Range Checking

Robert C. Seacord, Software Engineering Institute [vita[1]]
Dan Plakosh, Software Engineering Institute [vita[2]]

2005-09-27

Integer range checking, if implemented correctly, can eliminate vulnerabilities resulting from integer overflow, truncation, and sign errors.

Development Context

Integer operations

Technology Context

C, C++, IA-32, Win32, UNIX

Attacks

Attacker executes arbitrary code on machine with permissions of compromised process or changes the behavior of the program.

Risk

Integers in C and C++ are susceptible to overflow, sign, and truncation errors that can lead to exploitable vulnerabilities.

Description

The burden for integer range checking in C and C++ is placed squarely on the programmer's shoulders. Sometimes it's relatively easy, sometimes it's not. It's relatively easy, for example, to check an integer value to make sure it is within the proper range before using it to index an array. An out-of range value (for example, a negative integer) can be used by an attacker to bypass a check on the upper bound of an array and cause a buffer overflow. Figure 1 illustrates an example that uses both implicit type checking and explicit range checks to prevent an out-of-range integer value from causing a potentially exploitable vulnerability.[13] The implicit type check results from the declaration, on line 3, of len as an unsigned integer. In general, it is a good idea to use unsigned types for indices, sizes, and loop counters that should never have negative values. The `memcpy()` call on line 7 is also protected by an explicit range check on line 6 that tests both the upper and lower bounds.

1. daisy:274 (Seacord, Robert C.)
2. daisy:268 (Plakosh, Daniel)
13. For elucidatory purposes, Figure 1 does not include checks to ensure that the correct number of arguments is passed to the function. As a result, this program as listed is susceptible to crashes if argc < 3.

Figure 1. Implementation with implicit type and explicit range checking

```
 1. #define BUFF_SIZE 10
 2. int main(int argc, char* argv[]) {
 3.   unsigned int len;
 4.   char buf[BUFF_SIZE];
 5.   len = atoi(argv[1]);
 6.   if ((0 < len) && (len < BUFF_SIZE) ) {
 7.     memcpy(buf, argv[2], len);
 8.   }
 9.   else
10.     printf("Error copying data.\n");
11. }
```

Declaring len to be an unsigned integer is insufficient for range restriction because it only restricts the range from 0..MAX_INT. The range check on line 6 is sufficient to ensure that no out-of-range values are passed to `memcpy()` as long as both the upper and lower bounds are checked. Using both the implicit and explicit checks may be redundant, but we recommend this practice as “healthy paranoia.”

In other cases, type range checking is more complicated. For example, if x is assigned the product of a, b, and c, it is necessary to limit the range of a, b, and c so that the value of x cannot exceed the range of values for whichever integer type x is declared. As integer variables are operated on multiple times in combination with other integer values, it becomes increasingly difficult to ensure that an integer type range error does not occur.

While this problem is difficult to solve, there are valid software engineering techniques that can help. First, all external inputs should be evaluated to determine whether there are identifiable upper and lower bounds. If so, these limits should be enforced by the interface. Anything that can be done to limit the input of excessively large or small integers should help prevent overflow and other type range errors. Furthermore, it is easier to find and correct input problems than it is to trace internal errors back to faulty inputs.

Second, typographic conventions can be used in the code to distinguish constants from variables. They can even be used to distinguish externally influenced variables from locally used variables with well-defined ranges.

Third, strong typing[21] should be used so that the compiler can be more effective in identifying range problems.

####

###	########
Copyright Holder	Pearson Education

21. daisy:318 (Strong Typing)

####

###	########
is-content-area-overview	false
Content Areas	Knowledge/Coding Practices
SDLC Relevance	Implementation
Workflow State	Publishable